SAMGRID WEB SERVICES

S. Veseli, FNAL, Batavia, IL 60510, USA

Abstract

SAMGrid is a distributed (CORBA-based) HEP data handling system presently used by three running experiments at Fermilab: D0, CDF and MINOS. User access to the SAMGrid services is provided via Python and C++ client APIs, which handle the low-level CORBA calls. Although the use of SAMGrid API's is fairly straightforward and very well documented, in practice SAMGrid users are facing numerous installation and configuration issues.

SAMGrid Web Services have been designed to allow easy access to the system by using standard web service technologies and protocols (SOAP/XML, HTTP). In addition to hiding complexity of the system from users, these services eliminate the need for the proprietary CORBA-based clients, and also significantly simplify client installation and configuration.

We present here the architecture and design of the SAMGrid Web Services, and describe the functionality that they currently offer. In particular, we discuss various dataset and cataloguing services, as well as cover in more details the techniques used for delivering data files to end users. We also discuss service testing and performance measurements, deployment plans, as well as plans for future development.

INTRODUCTION

SAMGrid [1] is a general data handling system designed to work for high energy physics experiments with peta-byte sized datasets and widely distributed production and analysis facilities. After several years of operations[*], SAMGrid has evolved to be both robust and fault tolerant system, with a wide range of services offered to its users. Even though at this point the SAMGrid software is in a fairly mature state, there is still room for improvement, most notably in the areas of monitoring, software installation, configuration, and client access.

In this paper we describe efforts to offer additional means of accessing the system by implementing a web service[†] layer on top of the existing SAMGrid infrastructure. We present the architecture of the SAMGrid Web Services [3], and describe their present functionality, such as various cataloguing and dataset services, as well as mechanisms for delivering data to end users. We discuss service testing and deployment plans, and outline possible directions for future development.

[*] The system is currently used by three running experiments at Fermilab: D0, CDF and MINOS.

[†] For specifications of various web service technologies and protocols see Ref. [2].

EXISTING SAMGRID CLIENT ACCESS

As already mentioned, SAMGrid offers a wide variety of services, such as job management, data management, data transfer and storage, process accounting, etc. All services which require database access[‡] are encapsulated within the SAMGrid *DB Server* [4]. Examples of those are various cataloguing and dataset services. On the other hand, services involving data management, transfer and storage are provided by a set of SAMGrid *Station Servers*[§].

The primary means of accessing the SAMGrid system is via Python [5] and C++ [6] client APIs, but some of the cataloguing and dataset services are also provided by CGI scripts [7] and Java Servlets [8,9]. Python API incorporates all of the SAMGrid functionality, including various administrative and monitoring interfaces. It is distributed as a frozen binary with accompanying necessary shared object libraries.[**] This technique has the advantage that users have full access to all of the SAMGrid interfaces, as well as to the standard Python modules, without worrying about possible compatibility issues related to a specific version of Python installed on a given system. It is also worth mentioning that SAMGrid command line interfaces are built on top of the Python API. On the other hand, SAMGrid C++ API has much less functionality and is targeted for use in experiments' C++ reconstruction and analysis software. Similar to the Python API, the C++ API is distributed as self-contained set of libraries and header files.

Since the SAMGrid software uses CORBA [11] for communication between different components of the system, both APIs handle the low level CORBA calls, as well as utilize similar techniques for hiding generated CORBA structs from users by wrapping them into corresponding Python/C++ classes. The fact that the CORBA knowledge is not required makes the usage of SAMGrid APIs fairly straightforward.

Nevertheless, SAMGrid users are still facing non-trivial software installation and configuration issues. Besides an obvious issue of the SAMGrid API distributions being tied to a particular operating system[††], the client software also has to be properly configured before usage. At

[‡] The SAMGrid system relies on a centralized Oracle RDBMS.

[§] *Station* denotes a particular set of hardware resources that are managed by SAMGrid servers. End users request a set of files by submitting a SAMGrid *project* to one of the SAMGrid stations. Their applications are served input files by the *project manager* (one of the station servers).

[**] SAMGrid distribution of the Python API utilizes cx_Freeze utilities [10].

[††] At the moment Python API is distributed for Linux and SunOS, while the C++ API is distributed for Linux only.

minimum, one has to know the stringified IOR[‡‡] for the SAMGrid naming service, as well as the name of the appropriate production DB Server. In the worst case scenario, which involves retrieving files, one also has to install and configure various file transfer utilities used by SAMGrid, as well as worry about possible firewall problems. These issues usually do not represent a major obstacle for SAMGrid access from machines or clusters on which the software was installed and configured by the designated SAMGrid administrators. However, they make it difficult for regular users to setup their desktops or laptops for accessing SAMGrid via the distributed APIs.

SAMGRID WEB SERVICES

Service Architecture and Implementation

In order to rectify this problem, we have decided to implement a web service layer on top of the existing SAMGrid infrastructure (see Figure 1). The basic architecture is rather simple: the web service container receives client SOAP requests, and translates those into corresponding CORBA calls using SAMGrid Python API. Once the result is received, SOAP response is generated and sent back to the client. The web service approach hides complexity of the system from users, as all of the necessary software installation and configuration is done behind the scenes on the machine hosting the service. On the client side the URL of the service description file (WSDL) is the only thing needed.

We have chosen Python as the implementation language for the SAMGrid Web Services. This allowed us to easily utilize the existing functionality in the SAMGrid Python API, which considerably increased speed of the service development. We also decided to use SOAPpy [12], a Python web service package. SOAPpy takes care of marshalling and un-marshalling SOAP messages, as well as of processing WSDL files. In addition to that, it comes with a reliable threaded service container, which greatly simplifies the service deployment.

SAMGrid WSDL interfaces have been designed to closely match the CORBA IDL interfaces provided by the different components of the system. We have organized the implemented WSDL interfaces in the following web services [2]:

- Dataset Service
- Dimension Service
- DataFile Service
- Station Service
- Project Service

Although the functionality provided so far in the above services is not complete (for example, no administrative interfaces have been implemented), it is sufficient for most of the regular SAMGrid usage: retrieving file metadata and replica locations, searching for files based on specified constraints, defining datasets, and retrieving individual files or the whole datasets. In terms of user functionality, the most notable missing feature is the ability to declare and store new files into the system.

[‡‡] IOR (or interoperable object reference), is a reference to a CORBA object.

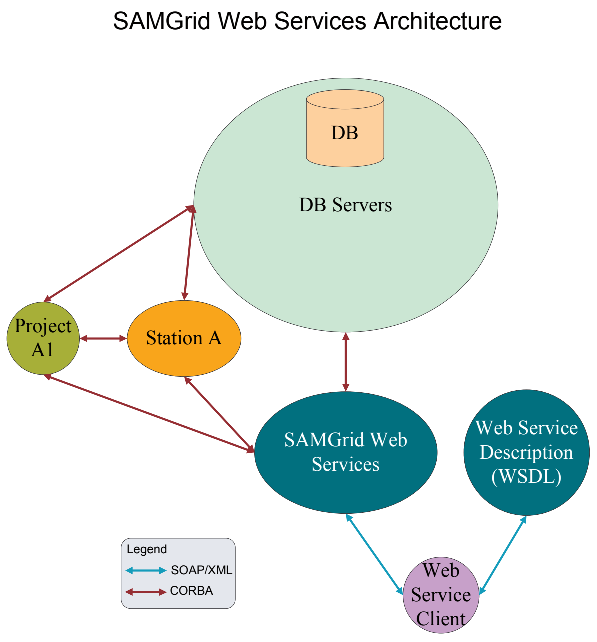

Figure 1: SAMGrid Web Services DB Architecture.

File Delivery Service

As already mentioned, in most cases implementation of the WSDL interfaces has been fairly straightforward and mainly involved translation between CORBA and SOAP. However, the delivery of data files was more complex, on both server and client side. Since the technique of transferring files as MIME attachments to SOAP response messages is limited to small files[§§], we have established a protocol for retrieving data in small chunks.

User who wants a set of files delivered to his/her machine sends initial SOAP request to the Station Web Service for establishing a file or dataset stream. This requests contains the information (e.g., set of file metadata constraints) that can be resolved into an explicit list of files. The Station Web Service verifies the request and sends back a response containing the stream id, as well as information relevant to the data that was requested, such as the number and of files and the size of the entire dataset. At the same time, the service makes a request for file delivery to one of the designated SAMGrid station servers. As files arrive in its cache, the station server sends notifications to the Web Service CORBA listener, and the Web service retrieves them into its own cache area. Once they receive the initial response, clients use the assigned file stream id to request data in

[§§] Our tests indicated that including binary SOAP attachments larger than about 100MB was not practical.

chunks, which have typical size of 1-10MB. Every chunk of data is accompanied by information (such as the chunk number and size, file checksum, etc.) needed for client to easily assemble the file and verify its contents.

The advantages of the this technique are obvious: there is no limit on the size of dataset that can be transferred, the amount of encoded binary data that is transferred for each request is small and easily handled by both web service and its client, etc. On the other hand, some care has to be taken on the client side so that files are assembled correctly.

PERFORMANCE MEASUREMENTS

For testing and performance measurements we used the D0 development environment. The Oracle database utilized an 8-CPU (400MHz UltraSPARC II processors) Sun machine with 4GB of memory, while the DB Server was running on a 4-CPU Linux machine (2.4GHz Xeon processors) with 3.5GB of memory. The web services were deployed on a dual Athlon MP 2000+ Linux machine with 1GB of memory, and a small cluster of similar machines was utilized for our test clients. During our testing there was no other significant database activity that could considerably affect our results.

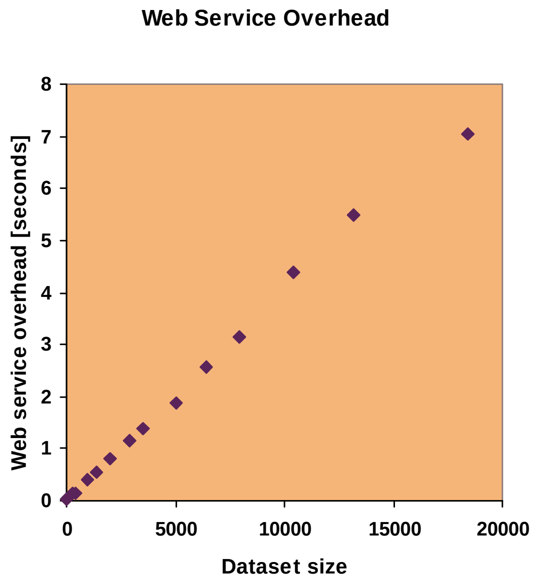

Figure 2: Web Service overhead as a function of the returned dataset size. The numbers shown correspond to about 12% of the total response time.

In order to understand the overhead associated with the additional SOAP call that has to be made, as well as with translating results returned by the DB Server into the corresponding SOAP/XML struct, we have performed a series of identical back-to-back SOAP queries, and measured the time between sequential DB Server calls which Web Service made on behalf of its client. In Figure 2 we show the average web service overhead as a function of the returned dataset size (i.e., the number of file names in the returned SOAP/XML list). The results shown were obtained with a single Web Service client and 10 sequential calls for each dataset. As expected, the time necessary for additional SOAP processing grows linearly with the size of the SOAP message. The numbers shown in Figure 2 correspond to about 12% of the total response time. Since the size of the returned SOAP struct is directly proportional to the size of the corresponding CORBA struct, the relative overhead time does not change with the type of the query used. However, speed of the machine hosting the web service, as well as of the machine running the client, plays a major role for time required to process SOAP messages. We also note that usually the query response time depends on a number of factors like the ongoing database activity, machine and network load, etc., so that in most cases the observed 12% Web Service overhead would fall within variation of the direct DB Server query timing. For example, in the case of a query returning a list of about 2500 files, response times for 10 direct DB Server calls were in the range of 8.78 to 9.82 seconds, with the average of 9.24 seconds. The same query against the Web Service had response times between 9.04 and 9.88 seconds, with the average of 9.32 seconds for 10 calls.

Our second series of tests had a goal of understanding and comparing the Web Service and the DB Server performance under load of a number of simultaneous clients. The DB Server was configured to use 5 connections to the database. Each client performed a set of queries, such as retrieving file metadata, retrieving a list of file locations, or a list of files in a given dataset. The same set of queries (unit of client work) was repeated 10 times. In Figure 3 we displayed our results for the average service response time per unit of client work as a function of the number of simultaneous clients. For a single client, the effects of SOAP overhead are clearly visible. In the case of two clients, the average response time goes down by a factor of two for both DB Server and Web Service. This is easily explained by the fact that the DB Server had multiple connections to the database, and therefore all queries could be executed in parallel. For three simultaneous clients the DB Server performance slightly deteriorates, which is an effect of saturating the number of available database connections.[***] However, it is interesting to note that the Web Service performance did not deteriorate as much, and for more than three clients the Web Service using the DB Server on behalf of its clients actually performed slightly better than the DB Server alone. This behaviour is a result of the DB Server architecture: the pool of available database connections is shared amongst all DB server proxies, and each DB Server proxy has to acquire the database connection before it can execute its query. All of the DB server direct

[***] Since each unit of client work in our test actually required two different DB Server proxies, and each DB Server proxy requires a database connection, 5 available database connections were saturated with only three clients.

clients have dedicated proxies. On the other hand, the Web Service maintains a pool of DB server proxies, which are shared amongst all if its clients. Since this mode of operation results in more of the DB Server proxy re-use, it also carries less overhead related to the database connection context switch.

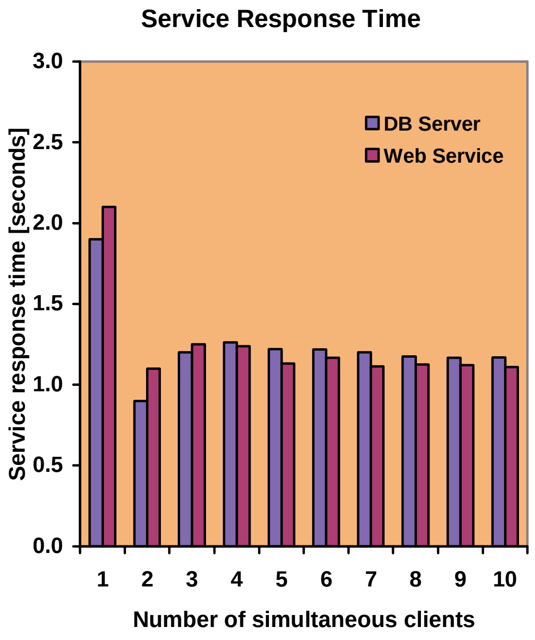

Figure 3: Average service response time per client unit of work (set of several different queries) as a function of the number of simultaneous clients.

We have repeated the load test using DB Servers running on different machines. Although the overall scale was different (more in favour of the Web Service for less powerful machine hosting the DB Server), the trend was always the same as the one shown on the Figure 3. It is also worth noting that we performed the same test with 100 simultaneous clients, and we have not experienced any problems with the Web Service (SOAPpy) container, nor we have seen any performance degradation. The average service response time per unit of work was 1.12 seconds for the Web Service, and 1.16 seconds for the DB Server alone.

In addition to the above tests, we have also measured performance of our file delivery service. Mostly due to encoding needed to make files suitable for transfer over HTTP, we observed data transfer rates of about factor of 2 slower than using the secure copy protocol, and up to 5 times slower than using the kerberized rcp.

FUTURE DEVELOPMENT

Although the current functionality offered by the SAMGrid Web Services is sufficient for most of the regular system usage, several important pieces are still missing. Most notably, the ability for users to declare and store new files into the system is not there. In addition to the services that have to be added, some work is also needed to improve performance and insure reliability of the file transfer service.

CONCLUSIONS

In this paper we have described the architecture, design and implementation of the new SAMGrid Web Services. These services offer additional means of accessing the system without complex installation and configuration of the standard SAMGrid software.

We have also discussed service testing and performance measurements. Our results show very good performance under various load conditions and with different usage patterns.

At this time the SAMGrid Web Services have been deployed in production for MINOS [13], and are being tested for use at D0.

ACKNOWLEDGEMENTS

We would like to thank Fermilab Computing Division for its ongoing support of the SAMGrid project. We would also like to thank everyone at D0, CDF and MINOS who has contributed to SAMGrid. This project is sponsored by DOE contract No. DE-AC02-76CH03000.

REFERENCES

[1] http://projects.fnal.gov/samgrid
[2] http://www.w3.org
[3] http://d0db.fnal.gov/sam_web_services
[4] L. Loebel-Carpenter et al., "The SAMGrid Database Server Component: Its Upgraded Infrastructure and the Future Development Plan", Proceedings of the Computing in High-Energy Physics (CHEP '04), Interlaken, Switzerland, 27 Sep - 1 Oct 2004.
[5] http://d0db.fnal.gov/sam_pyapi
[6] http://d0db.fnal.gov/sam_cpp_api
[7] http://d0db-prd.fnal.gov/sam_data_browsing
[8] http://dbb.fnal.gov:8520/cdfr2
(CDF Database Browser)
[9] http://d0db-prd.fnal.gov:19655/sam_dataset_editor
(SAMGrid Dataset Editor)
[10] http://sourceforge.net/projects/cx-freeze
[11] http://www.omg.org
[12] http://sourceforge.net/projects/pywebsvcs
[13] A. Kreymer et al., "Lightweight deployment of the SAM grid data handling system to new experiments", this Conference Proceedings.